Scalable Service Interface

9 February 2017

Andrew Hanushevsky

SLAC
NATIONAL ACCELERATOR LABORATORY

1 Introduction

This document describes the **S**calable **S**ervice **I**nterface (**SSI**) component of the **XRootD** framework. The **SSI** is a multi-threaded **XRootD** plug-in that implements a request-response framework. Using this framework you can write client applications that issue arbitrary requests to a server that executes the request and then responds with a result. Since the framework is based on **XRootD**, all of the base features available to any **XRootD** plug-in are available to be used by **SSI**. This includes server clustering which allows you to cluster an unlimited number of servers to achieve horizontal scaling.

Additionally, all of the actions taken by the framework that may involve a delay are asynchronous using a callback mechanism for serialization. That is, when an action is taken that may involve a delay, it is launched in the background and the requestor is told that the results of the action will be made available when a requestor-provided callback is invoked. This minimizes the number of threads needed to implement a service as the thread requesting an action is free to do other work while the action is executed. This maximizes scaling within a process.

The **SSI** framework implements a remote object execution model. In this model, actions taken by the client, such as object creation, are symmetrically performed at the server. In this model client actions appear, to the extent possible, to occur locally while the actual execution is remote. This simplifies writing complex client-server interactions and normally very little framework code is needed to implement a request/multiple-response service.

While **SSI** is implemented using **XRootD** protocol, the **SSI** related classes do not expose any **XRootD** dependencies. Thus, it is possible to port the framework to most other protocols without impacting existing service applications.

1.1 Key Concepts

The **SSI** is an object based framework using the following key concepts:

- a service provider,
- one or more services,
- one or more resources,
- one or more requests, and
- request responders.

A service provider is responsible for creating a service. When a service is created, the initial point of contact (i.e. host and port) must be specified. The initial point of a contact can be a single node or the head node of a node cluster. Hence, a service is tied to one or more nodes hosting the service. There can be any number of services, each with its own unique set of resources. Usually, there is only one type of service.

A resource is an abstract name of some capability offered by a service. It is up to the implementation to assign names to particular resources available via a service. The resource simply identifies what future requests will be using along with possible restrictions.

Once a service is created, one or more requests using the resource can be executed within the context of the service.

The request responders are a server-side concept needed to send a response in the context of a specific request. A response can be:

- fixed amount of arbitrary data,
- an arbitrary amount of streamed data,
- a file, or
- an error message.

The server side service implementation determines the appropriate response.

In its simplest form, resources can be provided by a single server. That server is responsible for processing requests and providing responses. The next few sections assume such a setup as it is simpler to explain the object interactions using a single server. A more complex scheme is a setup of several servers with a mixture of resources, some replicated and some not, configured in a cluster. The concepts presented in the simple case apply to clustered servers with some additional classes to manage the cluster. This is covered later.

1.2 Key Classes

The following table outlines the classes used on the client as well as the server. In most cases these are the same. It is best to read the detail header file comments on how to use the classes. Examples shown in this document do not necessarily show all possible options.

Class	Header	Client	Server	Comments
XrdSsiCluster	XrdSsiCluster.hh	n/a	opt	Effects cluster actions
XrdSsiEntity	XrdSsiEntity.hh	n/a	opt	Describes an incoming client
XrdSsiErrInfo	XrdSsiErrInfo.hh	Y	Y	Describes any error
XrdSsiLogger	XrdSsiLogger.hh	n/a	opt	Routes log messages
XrdSsiProvider	XrdSsiProvider.hh	Y	Y	Service factory
XrdSsiRequest	XrdSsiRequest.hh	Y	Y	Embodies a client request
XrdSsiResource	XrdSsiResource.hh	Y	Y	Describes a resource
XrdSsiRespInfo	XrdSsiRespInfo.hh	Y	Y	Actual response
XrdSsiResponder	XrdSsiResponder.hh	n/a	Y	Responds to a request
XrdSsiService	XrdSsiService.hh	Y	Y	Describes a service
XrdSsiStream	XrdSsiStream.hh	opt	opt	Streams response data

In the table above, “n/a” indicates the class is not applicable in the given context and “opt” indicates that the class is optional in the given context (i.e. it may not be needed, depending on the service being provided).

Implementing a client that uses an existing service provider is the simplest way to start. The following sections provide a client-side example and show how actions on the client are reflected on a server. This provides a clear explanation on how to use these classes on a client and can be used to guide a remote service implementation. This will be augmented in following sections that guide you through implementing a simple remote service.

1.3 Callbacks, Threads, and Mutexes

Programming in a callback-centric environment, while not difficult, can be challenging. This is because the callback is not only asynchronous, meaning it can happen at any time, there is no guarantee whether or not a separate thread is used to effect the callback. For instance, when a thread calls an **SSI** method that method may perform an immediate callback using the calling thread. If the caller is holding a mutex that must also be locked in the callback method, a deadlock will occur.

To avoid such scenarios do one of the following:

- do not hold any mutexes when calling any **SSI** method that responds with a callback, or
- use recursive mutexes.

Following this simple formula will avoid hours of debugging time.

1.4 Security

The full **XRootD** authentication suite is available for use with the **SSI** framework. By default, no authentication is configured. Refer to the "**XRootD** *Security Configuration Reference*" enable authentication as well as request verification to prevent man-in-the-middle attacks. Authorization possibilities are detailed in the authorization section of this document.

2 The SSI Client

In a client application, the **SSI** framework already provides a service that allows the client to use remote service providers. This service merely coordinates the interactions between the client and the remote server and should be viewed as simply an extension of the remote service provider itself. In order to get access to client-side service coordinator the application must link with **libXrdSsiLib.so** that contains the framework implementation.

2.1 Step 1: Obtain a Service Provider

The first step is to obtain a service object that corresponds to a service provider. This is done using the built-in provider object. The following code fragment shows how to do this.

```
#include "XrdSsi/XrdSsiProvider.hh"
#include "XrdSsi/XrdSsiService.hh"

extern XrdSsiProvider *XrdSsiProviderClient;
       XrdSsiErrInfo   eInfo;
       XrdSsiService  *theSP;
       std::string     contact("somehost:1234");

theSP = XrdSsiProviderClient->GetService(XrdSsiErrInfo eInfo,
                                         contact);

if (!theSP) {…} // eInfo has the reason for the failure
```

Here, contact is a string that holds the location of the service (e.g. *hostname*:*port*) such as "somehost:1234". This is the initial point of contact for future requests. The initial point of contact is not validated at this time. However, other errors may occur that cause the call to return a nil pointer. If a nil pointer is returned the **eInfo** object contains an error message along with an **errno** value describing why the call failed.

At this point the client is able to execute requests using the returned service object.

2.2 Step 2: Define a Resource

The next step is to instnatiate a resource. Resources are instantiated using the **XrdSsiResource** class (see the **XrdSsiResource.hh** header file). The resource object is handed off to the service obtained in step 1 along with a request object. A simple resource definition is shown below.

```
// Define a simple resource call "/mysql".
//
XrdSsiResource theResource("/mysql");
```

2.2.1 Resource Names

While resource names are arbitrary, by default they must start with a slash. It is possible to configure an **XRootD** server to accept names without a leading slash. The configuration options are covered under configuring resource names.

2.2.2 Reusable Resources

A reusable resource is one which can be used to bypass request resource setup overhead for subsequent requests using that resource. Reusable resources make sense when **SSI** servers are not clustered or, if clustered, resources are neither replicated nor migrated from server to server. Essentially, a reusable resource represents a service context for all requests that use that resource. Because reusable resources are cached, resource setup occurs only once, avoiding setup overhead.

You designate a resource as reusable by setting **XrdSsiResource::Reusable** in the **XrdSsiResource::rOpts** member. You control the lifetime of a reusable resource using the **XrdSsiResource::Discard** flag; which may also be set in **rOpts**. When **ProcessRequest()** sees that *either* option is set, it performs the following actions:

- A resource key is created by concatenating **XrdSsiResource::rUser** with **XrdSsiResource::rName**.
- A resource cache lookup is done using the key.
- If the resource is found in the cache but **Discard** has been set, it is removed from the cache.
- If the resource was either not found or discarded, a new resource context is created.
- If the **Resuable** option is set, the new context is inserted into the cache.

You should see that this processing logic uses a cached resource context, a refreshed cached resource context, or a new resource context if neither flag is set. You set the **Discard** flag to either remove a cached resource or refresh it, typically after a permanent error is returned. You should use **Discard** sparingly. Do not specify both flags for every request as the setting would be equivalent to specifying neither flag except for adding additional overhead to each request.

Since only the **rUser** and **rName** determine whether or not resource context is reused, you cannot send new **rInfo** information to the server via a reusable resource as this information is relayed only when the resource context is created.

2.2.3 Resource Affinity

Resource affinity is only relevant when you cluster **SSI** servers. If you are not clustering **SSI** servers then you can ignore this section.

In a clustered environment, if more than one server can provide a resource, the **SSI** framework tries to load balance requests across all of the servers that provide a requested resource. For certain types of requests this may be a disadvantage. For instance, say the start-up cost for request of type X is high so it would make more sense to always run the same type of request on the same server where the original type X resource landed. Or say that the same kind of requests always run symbiotically better when they run together on the same server. In this case, all such requests should be scheduled together.

Resource affinity provides you a way of requesting that requests using a certain resource run on the same server, if at all possible. You specify resource affinity requirements in the **XrdSsiResource** object setting an enum **Affinity** value in the **XrdSsiResource:: affinity** member. The following table describes the possible choices in ascending strength.

Affinity	Meaning
Default	Use affinity configured for the cluster's redirector. This is the default.
None	The resource has no affinity. Any server can be selected.
Weak	Run requests using this resource on same server if doing so does not involve a scheduling delay. Otherwise, any server will do
Strong	Run requests using this resource on same server even if doing so involves a scheduling delay.
Strict	Always run requests using this resource on same server regardless of any overhead in doing so.

The cluster's redirector determines which server to select for a request based on the resource name. The assumption is that more than one server may have the resource. Consequently, the redirector must determine all of the servers that have the resource to consistently use only one of them. This may take some time. You control how much you are willing to wait by specifying the strength of the affinity you want. The stronger the affinity, the more likely the same server will always be used. Usually, delays only occur on the first use of the resource. So, a weak affinity may cause two or three requests to go to different servers but subsequent requests will normally go to the same server.

Once affinity is established, the cluster's redirector will use the same server for all requests using the resource as long as affinity is specified. Should that server fail, an alternate server is selected to be the point of affinity. That server remains the point of affinity for the resource even if the original server comes back.

If you have more than one redirector, you should configure them in load balance mode. This is the only way a cluster of redirectors can maintain consistent affinity for a particular resource.

Default resource affinity is specified as part of the **cmsd** configuration using the **cms.sched** directive. Refer to the "*Cluster Management Service Configuration Reference*" for more information.

2.2.4 Esoteric Resource Options

Additional resource options are available. Some only apply to clustered environments. The esoteric options are:

XrdSsiResource::hAvoid

This option only is meaningful in a clustered environment. It specifies a list of comma separated host names or IP addresses of servers you do not want to use when selecting a server to handle the request using the particular resource. Be aware, that if the resource is only available on one of the servers in the list, the request fails.

XrdSsiResource::rInfo

This option allows you to send additional out-of-band information to the server that will be executing the request. The information should be specified in **CGI** format (i.e. key=value[&key=value[…]]). This information is supplied to the server-side service in its corresponding request resource object. Note that restrictions apply for reusable resources.

XrdSsiResource::rUser

This is an arbitrary string that is meant to further identify the request. The **SSI** framework normally uses this information to tag log messages. It is also supplied to the server-side service in its corresponding request resource object.

2.3 Step 3: Derive a Request Class

The request object must be defined as a derivation of the abstract **XrdSsiRequest** class. This is because it also has an abstract callback method (i.e. **ProcessResponse()** and optionally **ProcessResponseData()**) as well as an abstract method for the framework to obtain the request data (i.e. **GetRequest()** and optionally, **RelRequestBuffer()** and **Alert()**) to be sent to the server via the service object. The following minimal code snippet shows an example of how to do that.

```
#include "XrdSsi/XrdSsiRequest.hh"

class myRequest : public XrdSsiRequest
{
public:

virtual char *GetRequest(int &dlen)
                        {dlen = reqBLen; return reqBuff;}

virtual bool  ProcessResponse(const XrdSsiErrInfo  &eInfo,
                              const XrdSsiRespInfo &rInfo);

virtual void  ProcessResponseData(char *buff, int blen,
                                  bool  last);

              myRequest(char *buff, int blen)
                       : reqBuff(buff), reqBLen(blen) {}
virtual      ~myRequest{} {}

private:
char         *reqBuff;
int           reqBLen;
};

bool myRequest::ProcessResponse(const XrdSsiErrInfo  &eInfo,
                                const XrdSsiRespInfo &rInfo);
{
   if (eInfo.hasError()) {…} // eInfo has the failure reason
      else {…}               // rInfo holds the response
   return true;
}

void myRequest::ProcessResponseData(char *buff, int blen,
                                    bool  last)
{…} // Example implementation shown later
```

Here we define the **GetRrequest()** method to simply return the pointer to the buffer holding the request data and the length of the request data. It is up to the implementation to create request data, save it in some manner, and provide it to the framework when **GetRequest()** is called. The optional **RelRequestBuffer()** method can be used to minimize memory usage as it is called once the framework no longer needs access to the request data (see the header file for details). How you generate the request data or supply it to the **GetRequest()** call is up to you. However, be aware that the thread used to initiate a request may be the same one used in the **GetRequest()** call and this may affect your implementation choice.

Since a request is asynchronously sent to a server via the service object, the **ProcessResponse()** callback method is used to inform the request object that the request completed or failed. At that point you are assured that a response is actually ready. This method is called on a new thread.

The optional **ProcessReesponseData()** is another callback method that is used in conjunction with the request's **GetReesponseData()** method or when the response is a data stream and you wish to asynchronously receive data via the stream. Most, but not all, applications will need to implement a **ProcessReesponseData()** method since a) it is more convenient to use **ProcessReesponseData()**, and b) scalable applications generally require that any large amount of data be asynchronously received. Hence, the example shows an implementation of **ProcessReesponseData()**.

The next step involves issuing a request. The details of handling the response are covered afterwards.

2.4 Step 4: Initiate a Request

Requests are always executed in the context of a service. Your requests need to correspond to what the service allows. Violating that will likely return an error. Based on the preceding steps, the following code fragment shows how to initiate a request.

```
#include "XrdSsi/XrdSsiRequest.hh"

XrdSsiService *servP;      // Obtained via XrdSsiProvider
XrdSsiRequest *theRequest; // Used for demonstration purposes
XrdSsiResource theResource("/mysql");

// Create a request object (upcast from your implementation)
//
theRequest = new myRequest(reqData, reqDLen);

// Hand off the request to the service object
//
servP->ProcessRequest(*theRequest, theResource);

// The thread is now free to do anything it wants!
```

Once you call **ProcessRequest()** you are effectively transferring control of the request object to the service object. This means

- you must not alter or delete the request data until after **RelRequestBuffer()** method is called. If you have not implemented this method, then you must not alter or delete the request data until the **ProcessResponse()** callback is invoked.
- You must not delete the request object before the request is finished (see finishing a request). This also applies to the request data if you did not implement a **RelRequestBuffer()** method.

Object ownership is used by the **SSI** framework to minimize data copying.

2.4.1 Initiating a Detached Request

Depending on the configuration, you may specify that a request run in a detached state. When a request is detached, it does not need a live **TCP** connection to execute. However, it may only stay in a detached state for a limited amount of time before being automatically cancelled.

You specify that you want the request to be detached by setting the detach time-to-live value in the request object to something greater than zero using the **XrdSSiRequest:: SetDetachTTL()** protected method. The method's argument specifies the maximum number of seconds that the request may stay detached. It must be set before you call **XrdSsiService::ProcessRequest()**. The default requires that the request run attached (i.e. detach time limit is zero).

Once the request is accepted for processing on a remote server, the **ProcessResponse()** callback is invoked with a response type of **isHandle**. The response is the handle you should use to attach to the request either on the same host or from anywhere else by passing the handle to the **XrdSsiService::Attach()** method. Therefore, you should save the handle and use it when you wish to reap the request's response as well as any pending alert messages.

The **Attach()** method requires that you not only give it the handle but also a request object to use. It need not be the same request object you used to initiate the request. It is needed in order to establish the set of callbacks and other methods required to process the response.

Be aware that detached requests may be prohibited or restricted, depending on the server's configuration.

2.5 Step 5: Process the response

After a server process the request it responds with the result. The response is always contained in the **XrdSsiRespInfo** object, a private member in the **XrdSsiRequest** object. That **XrdSsiRespInfo** object is passed to your implementation of **ProcessResponse()**. The following table lists all the possible response types you may receive relative to the object's member contents.

rType is	**buff**	**blen**	**eMsg**	**eNum**	**strmP**
isData	→ data buffer	buffer length	n/a	n/a	n/a
isError	n/a	n/a	→ message	errno	n/a
isHandle	→ handle	handle size	n/a	n/a	n/a
isStream	n/a	n/a	n/a	n/a	→ XrdSsiStream object

There are only three possible types of response that a client may see:

- a data buffer when **XrdSsiRespInfo::rType** is set to **isData**,
- an error when **XrdSsiRespInfo::rType** is set to **isError**,
- a detached request handle when **XrdSsiRespInfo::rType** is set to **isHandle**, or
- a data stream when **XrdSsiRespInfo::rType** is set to **isStream**.

When the response is an error, the error information is contained in the **XrdSsiErrInfo** passed to the callback. You can check **XrdSsiRespInfo::rType** or use the **XrdSsiErrInfo::HasError()** method to verify whether or not an error has occurred.

A special response type of **isHandle** is returned when a detached request has been successfully initiated. Handling this response type is covered under **isHandle** responses.

Otherwise, the only other possible response is actual data that the server sent. That data is provided to the client either directly via a buffer or via a stream object.
If the response type is **isData** then you should directly access the **RespInfo buff** and **blen** members to extract the response without copying data (i.e. using **GetResponseData()** for an **isData** response causes the response data to be copied). If the response type is **isStream** then you can use **GetResponseData()** or directly use the **XrdSsiStream** methods in the returned stream object via the **XrdSsiRespInfo**::**strmP** member (either approach is equally efficient).

That said, it is easier to use the **GetResponseData()** method, as shown below. It allows you to avoid dealing with the details of the response type. Should you wish to directly use the stream object, see the description of this object either in its header file or refer to the section explaining streams.

```
#include "XrdSsi/XrdSsiRequest.hh"

bool myRequest::ProcessResponse(const XrdSsiErrInfo  &eInfo,
                                const XrdSsiRespInfo &rInfo)
{static const int myBSize = 1024;

 if (eInfo.isOK()) {char *myBuff = malloc(myBSize);
                    GetResponseData(myBuff, myBSize);
                   }
    else {/* Handle error using the passed eInfo object */
          Finished(); // You can now do a "delete this"
         }
return true;
}
```

The previous code snippet shows a simple response handler. Notice that when the **eInfo.isOK()** returns false then the response is an error (i.e. **rInfo.rType == isError**). Otherwise, the response is data via a buffer or a stream. In the example, we allocate a data buffer of some arbitrary size and call **GetResponseData()** passing it the buffer and its size. The response data will be placed in the buffer. How you actually handle response data buffer is up to you. In any case, you should always return true.

Once response data is placed in your buffer the request object's **ProcessResponseData()** method is called. You need to implement this method as part of your request object. A sample implementation is shown below.

```
#include "XrdSsi/XrdSsiRequest.hh"

XrdSsiRequest::PRD_Xeq myRequest::ProcessResponseData
                       (char *myBuff, int myBLen, bool isLast)
{
static const int myBSize = 1024;

// Process the response data of length myBLen placed in myBuff
// If there is more data then get it, else free the buffer and
// indicate to the framework that we are done
//
if (!isLast) GetResponseData(myBuff, myBSize);
   else {free(myBuff);
         Finished(); //You can now do a "delete this"
        }
return XrdSsiRequest::PRD_Normal;
}
```

The **ProcessResponseData()** method is passed the original buffer you supplied to **GetResponseData()** as well as the amount of data placed in the buffer. If you want to reuse this buffer you will need to track its size in a better way than shown. The **isLast** argument is set to true to indicate that there is no more response data. Otherwise, the supplied buffer was not large enough to contain all of the data and you can call **GetResponseData()** again to obtain it.

Be aware that the data response may zero-length. This can happen when the response only contains metadata or your service has no relevant response data. See the section "Metadata-Only Response".

The examples show that after the response is handled the request's **Finished()** method is called. This is a critical call and is explained in the section titled "Finish or Cancel the Request". Keep in mind that once **Finished()** is called, you must not reference any **XrdSsiRespInfo** object members as these are deleted as part of finishing the request.

Additionally, **ProcessResponseData()** must return one of the **PRD_Xeq** enums defined in **XrdSsiRequest**. Typically, **PRD_Normal** is always returned to indicate that normal post-processing is desired. It is possible to return an indication that the callback is to be held. This is covered under Pacing Responses.

2.5.1 Obtaining Optional Metadata Ahead of the Response

The **SSI** framework allows a server to send metadata ahead of the response data. This metadata may be used for any purpose. For instance, the metadata may describe the response so that it can be handled in the most optimum way.

You obtain any sent metadata with the **XrdSsiRequest::GetMetadata()** method using the request object associated with the response. Typically, you should get metadata in the **ProcessResponse** callback method before you issue **XrdSsiRequest::GetResponseData()**.

The metadata, if any, is persistent until you call **XrdSsiRequest::Finished()** method. After that call you must not reference the buffer holding the metadata.

2.5.2 Handling a Metadata-Only Response

The **SSI** framework allows a server to send only metadata as a response (i.e. the response only consists of metadata). This is useful for sending short responses and avoiding additional client-server handshakes to determine that there is no actual response data other than the metadata. A metadata-only response is indicated when the response type is **isData** and the length of the data is zero.

There is no hard and fast rule in using metadata-only responses to avoid communication overhead. The **SSI** framework tries to avoid such overhead for relatively small data responses as well.

You obtain any sent metadata with the **XrdSsiRequest::GetMetadata()** method using the request object associated with the response. Typically, you should get metadata in the **ProcessResponse** callback method. You should *not* call **XrdSsiRequest::GetResponseData()** as there is no other response data to be received. A metadata-only response is indicated when **XrdSsiRespInfo::rType** is set to **isNil** (i.e. no response data is present).

The metadata, if any, is persistent until you call **XrdSsiRequest::Finished()** method. After that call you must not reference the buffer holding the metadata.

2.5.3 Handling an isHandle Response

You get a **isHandle** response only when a detached request was accepted for processing on a remote server. The handle is used to reattach the request to a request object. The handle is an ASCII null-terminated string whose length includes the null character. You should copy the handle and then call **Finished()** to cleanup processing. See the section on initiating detached requests on how to use the handle.

2.5.4 Fielding Alert Messages

This **SSI** framework allows a server to send one or more alert messages. The format and meaning of an alert is strictly what you define it to be. Generally, alert messages should be short notification but practically there really is no restriction on how you use alerts.

You should supply an implementation of **XrdSsiRequest::Alert()** virtual method in order to accept alerts. This method is called whenever a server send your request an alert message. The default implementation ignores alert messages.

The **Alert()** method is called with a X**rdSsiRespInfoMsg** object. The object encapsulates the alert message. Once you have received and processed an alert message you should call **XrdSsiRespInfoMsg::Recycle()** to release resources allocated to the message. Failure to do so creates a memory leak. Details on the **XrdSsiRespInfoMsg** class can be found in the public **XrdSsiRespInfo.hh** header file.

An example on an **Alert()** method implementation is shown below.

```
#include "XrdSsi/XrdSsiRequest.hh"

class myRequest : public XrdSsiRequest
{
public:
virtual void Alert(XrdSsiRespInfoMsg &aMsg)
                  {int   aMsgLen;
                   char *aMsgData = aMsg.GetMsg(aMsgLen);
                   // Process the alert then recycle the msg
                   aMsg.Recycle();
                  }
••• // The remainder of your class definition
};
```

2.5.5 Pacing Data Responses

If you need to delay handling response data (e.g. it is too large relative to other things that need to occur), the **SSI** framework allows you to postpone the **ProcessResponseData()** callback to a later time. You do this by returning

XrdSsiRequest::PRD_Hold
: The callback is placed in a global hold queue and releases the thread for other work. Responses in the queue are restarted upon request in FIFO order.

XrdSsiRequest::PRD_HoldLcl
: The callback is placed in a local queue associated with the request identifier passed to the **XrdSsiRequest** constructor when the request object was allocated. If there is no request identifier, the callback is placed in the global queue. Responses in the queue are restarted upon request in FIFO order.

Calling the static method **XrdSsiRequest::RestartDataResponse()** restarts one or more **ProcessResponseData()** callbacks. The method accepts the number callbacks to restart and an optional request identifier. If a request identifier is not specified, the global queue is used. Otherwise, the queue associated with the specified request identifier is used.

When the **ProcessResponseData()** callback is restarted it is called with the same arguments before it was suspended.

2.6 Step 6: Finish or Cancel the Request

Recall that when you passed your request object to the service's **ProcessRequest()** method, ownership of the object transferred to the service object and you were not allowed to delete the request object. The request's **Finished()** method (supplied by the framework) is used to regain control of the object. It may be called at any time after the return from **XrSsiService::ProcessRequest()** but it must be called and called only once. When you call **Finished()** before the response is fully processed (i.e. before your **ProcessResponse()** method is called or there is still outstanding response data) the request is considered to be cancelled. Good programming practice requires that you explicitly indicate cancellation by passing true as an argument to **Finished()**. When Finished is improperly called, it returns false to indicate that it is likely that your program has a logic error. Otherwise, it returns true.

```
#include "XrdSsi/XrdSsiRequest.hh"
•••
if (!Finished()) abort()
   else delete this;
•••
```

In the above code snipped, calling **Finished()** returns ownership of the request object back to the caller. At this point you can delete the request object. If your request objects are uniform you should consider reusing them to avoid repeated object allocations.

You should also be aware that when you use **Finished()** to cancel a request, the cancellation is not guaranteed to be reflected to the server. This occurs when, from the server's perspective, the request has indeed finished because all remaining response data is already in transit.

2.7 Deleting Client-Side Objects

The following rules apply to safely delete the client-side objects described in the previous sections.

2.7.1 XrdSsiService Object

The **XrdSsiService** object cannot be explicitly deleted. To delete this object you must call its **Stop()** method which deletes the object if it is safe to do so. A service object can only be deleted after all requests handed to the object have completed (i.e. **Finished()** has been called on each request). Any attempt to stop the service while there are still outstanding requests causes the **Stop()** method to return false and the object is not deleted.

2.7.2 XrdSsiResource Object

The resource object may only be deleted after **XrdSsiService::ProcessRequest()** method returns.

2.7.3 XrdSsiRequest Object

The request object can only be deleted after its **Finished()** method has been called. Unlike other objects, there is no safeguard from deleting the object prior to calling **Finished()**. Violating this rule is likely to cause an invalid memory reference.

3 The SSI Server

In a server, the **SSI** framework loads your service application as a plug-in. This means your code needs to be packaged as a dynamically loadable shared library.

Referencing the previous flow summary, it should become apparent that actions taken by the client are essentially recreated on the server. The following sections explain what the service application needs to provide.

3.1 Step 1: Define the Service Provider

Server-side services are provided via the **XrdSsiProvider** object. While the client-side has one that is built-in the server-side must define one. There are actually two types of processes:

- A **cmsd** process needs only to ascertain whether or not a resource is available on the node on which its **QueryResource()** method is called, and
- A **xrootd** process that actually provides the service via the **GetService()** method. The process still needs to be able to ascertain resource availability.

This section only describes the **xrootd** process. You need not have a **cmsd** process if you are not clustering your servers. Defining a resource lookup provider for a clustered environment is described in a subsequent section.

The service provider is used by **xrootd** to actually process client requests. The service provider object is pointed to by the global pointer **XrdSsiProviderServer** which you must define and set at library load time (i.e. it is a file level global static symbol).

When your library is loaded, the **XrdSsiProviderServer** symbol is located in the library. Initialization fails if the appropriate symbol cannot be found or it is a nil pointer.

The three methods that you must implement in your derived **XrdSsiProvider** object are:

- **GetService()**,
- **Init()**, and
- **QueryResource()**

The **QueryResource()** method is used to obtain the availability of a resource. This method may be called whenever the client asks for the resource status. The

following code snippet shows how you would typically define the provider pointer at file level.

```
#include "XrdSsi/XrdSsiProvider.hh"

class MyServerProvider : public XrdSsiProvider {•••};

XrdSsiProvider *XrdSsiProviderServer = new MyServerProvider;
```

Once the provider object is found, its **Init()** method is called. The method should initialize the object for its intended use. Subsequently, a one-time call is made to its **GetService()** method to obtain an instance of an **XrdSsiService** object.

The following page shows a sample derivation of an **XrdSsiProvider** class. Note that it implements two pure abstract methods: **GetService()** and **QueryResource()**.

```
#include "XrdSsi/XrdSsiProvider.hh"
#include "XrdSsi/XrdSsiService.hh"

class myServceProvider : public XrdSsiProvider
{
public:

// The GetService() method must supply a service object
//
XrdSsiService *GetService(XrdSsiErrInfo     &eInfo,
                          const std::string &contact,
                          unsigned int       oHold=256
                         );

// Init() is always called before any other method
//
bool           Init(XrdSsiLogger      *logP,
                    XrdSsiCluster     *clsP,
                    const std::string  cfgFn,
                    const std::string  parms,
                    int   argc, char **argv
                   ) {•••
                      initOK = true; // If all went well
                      return initOK;
                     }

// The QueryResource() method determines resource availability
//
XrdSsiProvider::
rStat          QueryResource(const char *rName,
                             const char *contact=0
                            );

               myServiceProvider() : initOK(false);
virtual       ~myServiceProvider() {}

private:

bool initOK;
};
```

3.1.1 XrdSsiProvider::Init() Arguments

The **Init()** method is called once after your shared library is loaded and before any other calls. The arguments that may be of interest are:

XrdSsiLogger **logP*

logP points to a generalized message routing object. You can use this object to include messages in the framework log file. Messages are automatically prefixed with a time stamp and, when relevant, the thread ID issuing he message. See **XrdSsiLogger.hh** include file for full details.

XrdSsiCluster **clsP*

clsP points to an object that is used to control server selection and manage resources that the server may have. It is only relevant in a clustered environment. Cluster management relative to this object is described later.

const std::string *cfgFn*

cfgFn holds the name of the configuration file used to initialize the server. You can add your own directives to this configuration file and parse them out during initialization. This allows you to have a single configuration file.

const std::string *parms*

parms, if not nil, are inline parameters specified on the directive that identified your shared library. Passing parameters in this way is purely optional.

int *argc*, **char** ***argv*

The *argc* and *argv* parameters have the same use as the ones passed to **main()**. However, these command line arguments have been filtered out to only contain arguments specific to your plug-in. Refer to the section on starting **SSI** daemons for more information.

3.1.2 XrdSsiProvider::QueryResource() Arguments and Return Values

Whenever an **SSI** daemon needs to know the status of a resource it calls **QueryResource()**. This is true of the **xrootd** and, in a clustered environment, the **cmsd**. The argument, *rname*, passed is exactly the same as specified by the client when it created a derived instance of the **XrdSsiResource** object in order to use named resource. The **QueryResource()** method should return one of three values:

XrdSsiProvider::notPresent
: The return value indicates that the resource does not exist.

XrdSsiProvider::isPresent
: The return value indicates that the resource exists.

XrdSsiProvider::isPending
: The return value indicates that the resource exists but is not immediately available. This is only useful in clustered environments where the resource may be immediately available on some other node.

The second argument, *contact*, is always nil for a server. It is only used by a client initiated query for a resource at a particular endpoint.

3.2 Step 2: Derive a Service Class

The provider object supplies a service object via the **GetService()** method. While the client has the option of obtaining multiple service objects, one for each service supplier end-point; that is meaningless for a particular server since, from the **SSI** framework's viewpoint, there can only be one service relative to an end-point. Regardless, you must supply such a service object. Below is a sample derivation of a **XrdSsiService** class.

```
#include "XrdSsi/XrdSsiProvider.hh"
#include "XrdSsi/XrdSsiService.hh"

class myService : public XrdSsiService
{
public:

virtual void ProcessRequest(XrdSsiRequest  &reqRef,
                            XrdSsiResource &resRef)

               {// Do whatever is needed by the request
                // relative to the specified resource.
               }

             myService() {…}
virtual     ~myService() {…}
};
```

You must implement the pure abstract method **ProcessRequest()**. This method is called when the client calls **ProcessRequest()** to hand off its request and resource objects. Essentially, the client's request and resource objects are transmitted to the server and passed into the service's **ProcessRequest()** method.

Two additional virtual methods:

- **Attach()**, optimize handling of detached requests and
- **Prepare()** to and perform preauthorization and resource optimization.

These are explained in the subsequent sections.

3.2.1 Performing Authorization via Prepare()

As part of the request setup, the **SSI** framework calls the **XrdSsiService::Prepare()** passing it the resource object ahead of executing any requests. The default implementation simply calls **XrdSsiProvider::QueryResource()** to ascertain whether or not the required resource is actually available on the server.

If you enabled authentication, the **Prepare()** call gives you a central place to preauthorize use of the resource by a particular client. Of course, you need to implement the appropriate authorization mechanism. This is a preauthorization step because it does not take into account any actual requests that may be issued by the client. So, you may need to further authorize based on the client's request when **ProcessRequest()** is called.

Since authorization is necessarily based on the client's identity, you need to enable **XRootD** authentication. When enabled, the client's identity and credentials are passed via the **XrdSsiResource::client** member. This is a pointer to the **XrdSsiEntity** object which provides authentication details. Refer to the **XrdSsiEntity.hh** header file for details.

The following code snippet shows how this can be done. Note that you need to override the virtual **XrdSsiService::Prepare()** method in your service class.

```
#include "XrdSsi/XrdSsiService.hh"

void myService::Prepare(      XrdSsiErrInfo  &eInfo,
                        const XrdSsiResource &rDesc)
{
// Perform any preauthorization, if needed. Authentication
// information is pointed to by rDesc.client; see header file
// XrdSsiEntity.hh for full details. If the client is allowed,
// return true. Otherwise, return false with an error.
//
   if (allowed) return true;
   eInfo.Set("Resource use disallowed!", EACCES);
   return false;
}
```

3.2.2 Performing Resource Optimization via Prepare()

Overriding the **XrdSsiService:: Prepare()** virtual method allows you to control and optimize the use of the resource. Upon return from **Prepare()** you may

- redirect the client to another end-point where it will attempt to issue the request, or
- delay the client for some specific amount of time before it attempts to issue the request at the same end-point.

The following code snippet shows how this can be done. Note that you need to override the virtual **XrdSsiService::Prepare()** method in your service class.

```
#include "XrdSsi/XrdSsiService.hh"

void myService::Prepare(      XrdSsiErrInfo  &eInfo,
                        const XrdSsiResource &rDesc)
{
static const int   delayTime = 10;
static const int   port      = 1094;
static const char *altHost   = "somehost.domain.edu";

// If we are too busy then delay the client
//
   if (toobusy)
      {eInfo.Set("Too busy, try later!", EBUSY, delayTime);
       return false;
      }

// If we need to send the client elsewhere tell the caller
//
   if (nothere)
      {eInfo.Set(altHost, EAGAIN, port);
       return false;
      }

// OK, we are set to go with future requests
//
   return true;
}
```

3.3 Step 3: Process a Request

In practice, processing requests is relatively straightforward. The service object's **ProcessRequests()** method is called with the request object describing the request along with the corresponding resource object. That is sufficient to get things going. Interaction with the request object is a bit more problematic. This is because asynchronous events can occur outside the scope of the service object's knowledge. The most significant event is request cancellation which can occur at any time. Cancellation may occur because the client requested it or because the client's **TCP** connection was lost.

The **SSI** framework provides a rather novel way of ensuring that your service is protected against any race conditions that may occur due to asynchronous events that affect the request. The **SSI** framework employs two mechanisms:

- The use of an inheritable class, **XrdSsiResponder**, that knows how to safely interact with the request object, and
- the notion of binding a request object to a particular **XrdSsiResponder** object.

The **XrdSsiResponder** class contains all the methods needed to interact with the request object (i.e. get the request, release storage, send alerts, and post a response). The object that you use to process and respond to requests should inherit the **XrdSsiResponder** class. This is usually some agent object that the service object creates for each request that it receives.

The object that inherits the **XrdSsiResponder** class must notify the **SSI** framework which request it is handling. In essence, a responder binds itself to a request for the duration of the request. All interactions with the request object then occur via methods in the **XrdSsiResponder** class. The **XrdSsiResponder::BindRequest()** should be used to establish a 1-to-1 relationship between the request object and the object responsible for the request. Once the relationship is established, you no longer need to keep a reference to the request object. The **SSI** framework keeps track of the request object for you.

The following code snippet shows an example of how a service object can hand off a request to an agent object. Since each request is provided a new thread, the code path that processes the request can continue to use that thread for request execution without the need of spawning an additional thread. In the example, the sample class **RequestProc** is used to execute the actual request. The class **myService**, illustrated in the previous section, is used to launch request processing.

```
#include "XrdSsi/XrdSsiResponder.hh"
#include "XrdSsi/XrdSsiService.hh"

class RequestProc : public XrdSsiResponder
{
public:

        void Execute() {int   reqLen;
                        char *reqData = GetRequest(reqLen);
                        // Parse the request
                        ReleaseRequestBuffer(); // Optional
                        // Perform the requested action
                       }

virtual void Finished(      XrdSsiRequest  *rqstP,
                      const XrdSsiRespInfo &rInfo,
                            bool            cancel=true)
                     {… // Reclaim any allocated resources}

             RequestProc() {}
virtual     ~RequestProc{} {}
};

void myService::ProcessRequest(XrdSsiRequest  &reqRef,
                               XrdSsiResource &resRef)
{
   RequestProc theProcessor;

// Bind the processor to the request. This works because the
// it inherited the BindRequest method from XrdSsiResponder.
//
   theProcessor.BindRequest(reqRef);

// Execute the request, upon return the processor is deleted
//
   theProcessor.Execute();

// Unbind the request from the responder (required)
//
   theProcessor.UnBindRequest();
}
```

The previous code is a very simple example. If request processing requires long waits, you may want to spawn a new disposable thread to process the request. This allows you to liberate it during waits and resume the request when the wait completes. Returning from **ProcessRequest()** does not change the fact that you own the request object up until **UnBindRequest()** is called. What you need to do when the inherited **Finished()** method is called is described under finishing a request.

The two key calls that you should always make are:

- **BindRequest()** to tell the **SSI** framework who will be responding to the request, and
- **UnBindRequest()** when you are done so that the **SSI** framework can reclaim the request object.

You should not call **BindRequest()** in your processor's constructor. This is because once a responder is bound to a request; all pending asynchronous events associated with that request are directed to the responder. Calling **BindRequest()** in a constructor may activate live code paths during the construction of the object; leading to unpredictable results.

You should not call **UnBindRequest()** prior to **Finished()** being called. If you do, **UnBindRequest()** ignores the call and returns false to indicate a calling sequence error. Otherwise, it returns true. **UnBindRequest()** also returns false if you call it more than one. Failing to call **UnBindRequest()** creates a memory leak. The **SSI** framework attempts to discover such leaks and logs them.

The **SSI** framework provides great flexibility in how you architect your service. That said, your service object *should not* inherit the **XrdSsiResponder** class. This is because the **SSI** framework obtains only one service object and uses it to handle all the requests it receives. The responder object can only be bound to a single request at a time making it impossible for a service object to respond to a request.

The following example assumes that requests come as an encoded stream of bytes (e.g. protocol buffers). This is probably the most typical scenario. So, the service object gets each request, decodes the bytes, and creates whatever parameters it needs to pass along to a specialized object that actually executes the particular request. This keeps request processing code relatively short and focused on a particular task that it needs to do. You can likely devise even simpler architectures to handle requests.

```
#include "XrdSsi/XrdSsiResponder.hh"
#include "XrdSsi/XrdSsiService.hh"

class RequestProc : public XrdSsiResponder
{
public:

virtual void Execute() = 0; // Defined by specialization

virtual void Finished(      XrdSsiRequest  *rqstP,
                      const XrdSsiRespInfo &rInfo,
                            bool            cancel=true)
                     {… // Reclaim any allocated resources}
};

// Derive specialized processors

class RequestProcX : public RequestProc
{
public:

virtual void Execute() {… // Perform the requested action }

             RequestProcX(request_parameters) {…}
virtual     ~RequestProcX{} {…}
};

•••// Additional processor derivations

// A handy error processor

class RequestProcError : public RequestProc
{
public:

virtual void Execute() {SetErrResponse(eMsg, eCode);}

             RequestProcError(const char *msg, int ecode)
                              : eMsg(msg), eCode(ecode) {}
virtual     ~RequestProcError{} {}

private:
const char *eMsg;
int         eCode;
};
```

```
void myService::ProcessRequest(XrdSsiRequest  &reqRef,
                               XrdSsiResource &resRef)
{
   int   reqLen;
   char *reqData = reqRef.GetRequest(reqLen)

// Decode the request and release the request buffer.
//
   •••
   reqRef.ReleaseRequestBuffer();

// Based on what is requested, create the appropriate request
// processing object and hand off execution.
//
   if (client_wants_x)
       {RequestProcX xProcessor(request_parameters);
        xProcessor.BindRequest(reqRef);
        xProcessor.Execute();
        xProcessor.UnBindRequest();
        return;
       }
   if (client_wants_y)
       {RequestProcY yProcessor(request_parameters);
        yProcessor.BindRequest(reqRef);
        yProcessor.Execute();
        yProcessor.UnBindRequest();
        return;
       }
   •••
// The request is invalid, respond with an error.
//
   RequestProcError errProc("Request is invalid!", EINVAL);
   errProc.BindRequest(reqRef);
   errProc.Execute();
   errProc.UnBindRequest();
}
```

Be aware that the **XrdSsiRequest** and **XrdSsiResource** objects become invalid once **UnBindRequest()** is called.

3.3.1 Detached Requests

By default, a request must maintain a live **TCP** connection to the client that issued the request during the time of its execution. If the **TCP** connect is lost, the request is automatically cancelled. A client may indicate that the request run detached. In this case, the client is given a handle to the request and the request does not require maintaining a live **TCP** connection. However, detached requests can only run in detached state for a limited time (the tome is specified by the client). If the request is not attached within its time to live window, it is automatically cancelled.

Normally, a service need not be concerned whether a request is running attached or detached as the **SSI** framework handles all of the nuances of such requests. However, if your service makes decisions on how to run a request based on whether it is attached or detached, then you should call **XrdSsiRequest::GetDetachTTL()**. If the returned value is non-zero then the request will enter the detached state upon return from **ProcessRequest()**. The returned value is the number of seconds it can remain in detached state.

Because the request does not enter the detached state until you return from **ProcessRequest()**, your service has the opportunity to reject such requests for any reason whatsoever by simply posting an error response prior to returning. This also allows you accept detached requests from some clients but not others and limit the total number of detached requests that are outstanding. Should you return without posting an error response while in **ProcessRequest()**, the request enters the detached state.

If you need notification that a request has been attached, then you should override the **XrdSsiService::Attach()** virtual method. When a client reattaches to the request, the **Attach()** method is called with the same request object that originally started the request and the resource object describing the resources associated with the request.

Be aware that the **SSI** framework allows a request to be attached by a client *different* from the one that originally started the request. For security purposes, you should reauthorize the client attaching to the request if you are performing any kind of request authorization. You can reject the attach request by setting an error message in the passed **XrdSsiErrInfo** object and returning false.

Finally, detached requests are, by default, disallowed. You must enable them using the **ssi.opts** directive in the configuration file.

An example of an **Attach()** method follows.

```
void myService::Attach(      XrdSsiErrInfo  &eInfo,
                       const std::string    &handle,
                             XrdSsiRequest  &reqRef,
                             XrdSsiResource *resp
                      )
{

// If needed, authorize that the client can attach to the
// resource and perhaps the actual request.
//
   •••

// If the client is not allowed to attach to the request,
// return an error otherwise succeed
//
   if (not_allowed)
      {eInfo.Set("Attach not permitted!", EACCES);
       return false;
      }
   return true;
}
```

3.4 Step 4: Post the response

After you process the request you must respond with a result. A response can be

- a data buffer along with the length,
- an error message with an errno value,
- an open file with a size whose contents are the response, or
- a stream object that supplies data.

The response is always contained in the **XrdSsiRespInfo** object, a private member in the **XrdSsiRequest** object. That **XrdSsiRespInfo** object is passed to your implementation of **Finished()** so that you can release the response resources after the response has been sent to the client. The following table lists all the possible response types you may receive relative to the object's member contents.

rType is	**buff**	**blen**	**eMsg**	**eNum**	**fdnum**	**fsize**	**strmP**
isData	→ data	data length	n/a	n/a			
isError			→ message	errno code			n/a
isFile	n/a	n/a	n/a	n/a	file descriptor	file size	n/a
isStream	n/a	n/a	n/a	n/a			→ XrdSsiStream object

All responses are set by calling the appropriate **SetResponse()** method that an object inherited from the **XrdSsiResponder** class. Once a response has been posted to a request it cannot be changed. Hence, only a single response is allowed. An error response is shown in the following code snippet

```
#include "XrdSsi/XrdSsiRequest.hh"
#include "XrdSsi/XrdSsiResponder.hh"

•••

// Tell the client the request was not valid
//
SetErrResponse("Invalid request!", EINVAL);

•••
```

3.4.1 Sending Optional Metadata Ahead of the Response

The **SSI** framework allows a server to send metadata ahead of the response data. This metadata may be used for any purpose. For instance, the metadata may describe the response so that the client can handle it in the most optimum way.

You send metadata with the **XrdSsiResponder::SetMetadata()** method using the inherited **XrdSsiResponder** class. The metadata must be set before you post a response calling **XrdSsiResponder::SetResponse()**. The maximum amount of metadata that may be sent is defined by **XrdSsiResponder:: MaxMetaDataSZ** constant member.

The metadata buffer must remain persistent until the **Finished()** method is called.

3.4.2 Sending Alerts

The **SSI** framework allows you to send asynchronous messages to the client's request object. This is done via **XrdSsiResponder::Alert()** which, in turn, calls **XrdSsiRequest::Alert()**. Both calls produce identical results, though using the responder's version provides better code consistency and is the preferred method.

Alert messages can be anything you want them to be. The **SSI** framework enforces the following rules

- alerts are sent in the order posted,
- all outstanding alerts are sent before the final response is sent (i.e. the one posted using a **SetResponse()** method),
- once a final response is posted, subsequent alert messages are not sent, and
- if a request is cancelled, all pending alerts are discarded.

In order to send an alert, you must encapsulate your message using an **XrdSsiRespInfoMsg** object. The object provides synchronization between posting an alert, sending the alert, and releasing storage after the alert was sent. It is defined in the **XrdSsiRespInfo.hh** header file.

The **XrdSsiRespInfoMsg** object needs to be inherited by whatever class you use to manage your alert message. The pure abstract class, **RecycleMsg()** must also be implemented. This method is called after the message is sent or when it is discarded. A parameter to the method tells you why it's being called.

The following, rather inefficient, code snippet shows a sample message handler class and the sending of an alert message. We presume the code that sends the alert has access to the methods in the **XrdSsiResponder** class.

```
#include "XrdSsi/XrdSsiRespInfo.hh"

class AlertMsg : public XrdSsiRespInfoMsg
{
public:

void RecycleMsg(bool sent=true) {delete this;}

     AlertMsg(const std::string &aMsg)
              : XrdSsiRespInfoMsg(CopyMsg(aMsg),aMsg.size()+1)
              {}
    ~AlertMsg() {}

private:
char *CopyMsg(std::string &msg)
              {theMsg = msg; return theMsg.c_str();}

std::string theMsg;
};

•••
std:;string myMsg = "I am still here!";

// Send an alert message
//
Alert(new AlertMsg(myMsg));
•••
```

The reason you need to pass a C-type version of the string in the above example is because it allows **XrdSsiRespInfoMsg** to implicitly handle any type of container. The reason we pass **aMsg.size()+1** as the size is because we want to send the null byte that terminates the C-type version of the string. The **CopyMsg()** method ensures that **XrdSsiRespInfoMsg** is passed a pointer to the copied the string.

You can determine whether or not the message was actually sent by checking the argument "**sent**". When true, the message was sent to the client. Otherwise, it was discarded because there was no reason to send the message. Be aware, the sending of a message to a client is no guarantee that the client actual receives it. The client must enable receiving alerts and must not have cancelled the request.

3.5 Step 7: Finish or Cancel the Request

This section is broken into two sections: 1) normal request completion, and 2) request cancellation. The reason is that the considerations of each event sufficiently differ to the extent that you may need to take different actions depending on whether a request completes normally or is cancelled.

3.5.1 Normal Request Completion

When the client-server exchange completes, the **SSI** framework calls the **Finished()** method in the object that has been bound to the request via a **BindRequest()** call. This allows the request responder to release all resources dedicated to the request. A client-server exchange completes normally when either all of the response data has been sent to the client.

In practice, client actions are asynchronous to the server. The server-side framework always calls **Finished()** when the full response has been sent to the client irrespective of the client calling its own request's **Finished()** method. This is done to speed up client-server interactions. While practically consistent, it is not logically consistent because while the data is in transit to the client the client may cancel the request at the same time. The client-side framework makes it appear that the request was cancelled even though the server-side framework will not be aware of it.

The following code snipped shows the typical processing sequence for the **Finished()** method. We assume the **requestProc** class has inherited the **XrdSsiResponder** class and overrides its pure abstract **Finished()** method.

```
#include "XrdSsi/XrdSsiResponder.hh"

void requestProc::Finished(      XrdSsiRequest  *rqstP,
                           const XrdSsiRespInfo &rInfo,
                                 bool            cancel)
{
•••
   if (cancel) {see section on request cancellation}

// Reclaim resources dedicated to the request and then tell
// caller the request object can be reclaimed.
•••
// Tell the framework we are done with the request object
//
   UnBindRequest();
}
```

3.5.2 Request Cancellation

A request is cancelled when any of the following happen:

- the client calls its request object's **Finished()** method before all of the response data has been sent,
- the **TCP** connection for an attached request is lost before all of the response data has been sent, or
- a detached request has not been attached within its timeout window.

When a request is cancelled, the **SSI** framework calls the **Finished()** method in the object that has been bound to the request via a **BindRequest()** call with the cancel argument set to true.

While it should be possible to fully clean-up resources dedicated to the request when **Finished()** is called *after* you post a final response (i.e. normal completion), it isn't necessarily the case for cancelled requests. This is because **Finished()** is called asynchronously to any processing that might be going on for the request being cancelled and the **SSI** framework doesn't know how you implemented that processing. Regardless of how you stop request processing, the fact that **Finished()** is called does not automatically reclaim the request object.

The discussion above should make it obvious that the request object cannot be automatically reclaimed because request processing may take some time to actually be cancelled. However, the **SSI** framework does ensure that the responder will not be able to successfully post a response once **Finished()** is called.

Once processing is stopped, **UnBindRequest()** should be called so that the **SSI** framework can reclaim the request object.

3.6 Deleting Server-Side Objects

The following rules apply to safely delete the server-side objects described in the previous sections.

3.6.1 XrdSsiService Object

Technically, the **XrdSsiService** object can only be deleted in its **Stop()** method. However **SSI** framework never tries to stop a service.

3.6.2 XrdSsiStream Object

The stream object may only be deleted after **Finished()** is called.

3.7 Overall Flow Summary

The table below shows the overall flow with a simple single data response. The flow is more complicated when the response is a data stream. Details on presented in the sections on streams.

Client Application	Client-Side SSI Framework		Server-Side SSI Framework	Server-Side Service
XrdSsiProvidorClient-> GettService()	Return XrdSsiService Object		XrdSsiProvidorServer-> GetService()	Return XrdSsiService object
Create XrdSsiRequest Call ProcessRequest() *async return*	Call BindRequest() Call GetRequest() write()	→	read() Create XrdSSiRequest Call ProcessRequest()	Call BindRequest() Call GetRequest() Perform service
	Call ProcessResponse()	←	Notify client of pending response	Call SetResponse() As data response
Call GetResponseData() *async return*	read() Call ProcessResponseData()	↔	write() Call Finished()	Cleanup Call UnBindRequest()
Call Finished()	Cleanup			

4 XrdSsiStream

The **XrdSsiStream** object is used to provide data from a streaming source. Examples of streaming sources are:

- Socket
- Incremental database query
- Character device

Essentially, a streaming source is any data source that provides data in an incremental fashion whose size is not easily determined or, if it is known, whose size is too large to conveniently handle in one interaction with the data source.

The **XrdSsiStream** is an abstract class whose creator must provide a concrete implementation suitable to the data source in question. Any data source can be used but the implementation must adhere to the invocation restrictions defined herein.

The **XrdSsiStream** object supports two types of streamlining modes:

1) Active streams where, when asked for data, the object supplies the buffer containing the data, and
2) Passive streams where, when asked for data, a buffer is supplied by the requestor that is to be filled with data.

The two types of streaming modes are described in subsequent sections.

4.1 Relationship to other classes

The **XrdSsiStream** object is only meaningful in the context of an **XrdSsiRequest** object to affect a response. Hence, the **XrdSsiStream** object *is* the response to a request. Subsequently, the requestor obtains actual response data using the provided **XrdSsiStream** object.

Recall that responses are posted to a request object using the **XrdSsiResponder::SetResponse()** method. This means that any object wishing to post a stream response must

a) inherit the **XrdSsiResponder** class, and
b) be bound to the request with a previous **XrdSsiResponder::BindRequest()** invocation.

The above two requirements allow an object to process and respond to a request without any extensive request bookkeeping and maintains a 1-to-1 relationship between a request and the object responsible for responding to the request.

4.2 Object Persistence

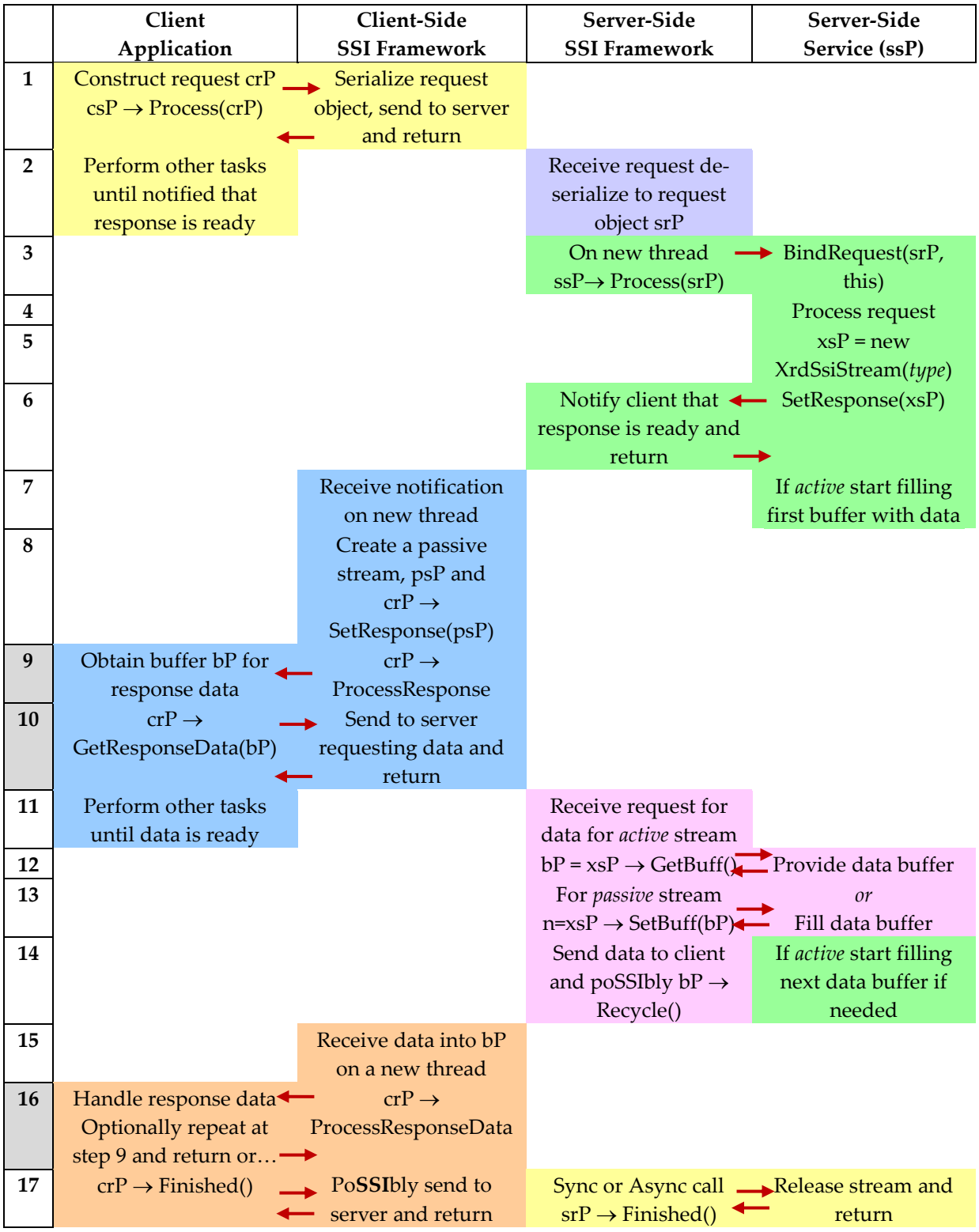
A stream object may not be deleted until all references to the object have been dropped. An explicit reference occurs when a stream object is exported via a call to **XrdSsiResponder::SetResponse()**. Only when **XrdSsiService**::**Finished()** is called with a pointer to the request object whose response holds the stream object can the creator of the stream object be assured that no **SSI** framework references exist to that stream object. Only then can the stream object be deleted.

The **XrdSsiResponder::Finished()** method is called when the **XrdSsiRequest::Finished()** is called indicating that the request has completed. A request can be marked as completed at any time even when not all of the stream data has been consumed. The stream implementation is responsible for reclaiming any unconsumed non-exported resources.

Be aware that active stream buffers that have been exported via the **XrdSsiStream::GetBuff()** method may only be reclaimed in the **XrdSsiStream::Buffer::Recycle()** method. This is because a reference to the buffer remains active until the buffer is recycled whether or not the request has been marked as finished.

4.3 Streaming Request-Response Sequence

The following diagram shows the sequence and time relationships between client actions and server actions relative to a stream. The sequence is the same whether an *active* or *passive* stream is used. The differences are described under the specific discussion of each type of stream. Red arrows show sequential calling sequences while block colors indicate individual threads. A detailed explanation follows.



Request-Response Sequence Using a Stream

Referencing the above diagram, the following steps occur:

1) The client constructs a request encapsulated by the **XrdSsiRequest** object pointed to by **crP** and passes it to the client-side service pointed to by **csP** for processing. The service object serializes the request object and sends it to its corresponding service object at some server. None of these actions block. If the request cannot be sent immediately it is queued and sent in the background.
2) When the call returns the application is free to perform any other required tasks. The request object will be informed that a response is ready when the server actually responds. In the mean time the server receives the request and recreates the request object pointed to by **srP**.
3) The **SSI** framework spawns a new thread to process the request. This is to avoid any timeout issues should the request take a long time to complete. It calls the server-side service object pointed to by **ssP** to process the request. In this example, the service object will be processing the request so it binds to the request to setup a 1-to-1 mapping between the request and the request processor.
4) The request is processed.
5) At this point the appropriate type of stream is created. If the results are either too large or cannot be immediately obtained an *active* stream may be created. Otherwise, a *passive* stream may be created. In either case **xsP** point to a stream of appropriate type.
6) Since the stream is the actual response, the service object calls **SetResponse()** pointing to the stream object. It can do so because it must have inherited the **XrdSsiResponder** class. As part of the **SetResponse()** call the **SSI** framework sends notification to the client that a response has been posted to the request object and returns. This is not blocking action.
7) At this point the service object, if using an *active* stream, is free to start filling a buffer with response data while the notification is in transit. At some point the client-side **SSI** framework receives the notification.
8) The **SSI** framework creates a passive stream, pointed to by **psP**, to relay the data from the server to the client. The framework matches the notification to the corresponding request object, **crP**. It then sets the passive stream as the response for the request using **SetResponse()**.
9) As part of the **SetResponse()** process, the **ProcessResponse()** method in the request object is called to notify the request that a response is ready. In this example, the request's **ProcessResponse()** method obtains a buffer to hold the response data.

10) It then calls the request object's **GetResponseData()** data method to fill the buffer with response data. Note that the **GetResponseData()** method is a convenience function that actually uses the passive stream to obtain the data freeing the application from having to deal with actual streams. The application could have used the passive stream directly if it wanted to. When the passive stream is asked to fill the buffer it sends a request for data to the server. Again, this is a non-blocking call.
11) Upon return from **GetResponseData()** the application is free to perform any other required tasks until it is informed that the data has arrived. In the mean time, the server has received the request for data.
12) *Active stream*: The **SSI** framework calls the *active* stream's **GetBuff()** method to obtain response data. The *active* stream simply hands over a buffer that contains some amount of data. Since this buffer has been exported to the framework it cannot be touched or freed until the framework indicates it is through with the buffer by calling the buffer's **Recycle()** method.
13) *Passive stream*: The **SSI** framework calls the *passive* stream's **SetBuff()** method to obtain response data. The framework supplies the buffer and its size and the stream must fill the buffer to the extent possible.
14) If enough data exists in the buffer, the data is sent to the client. Otherwise, the **SSI** framework will ask for (*active* step 12) another buffer or (*passive* step 13) another buffer fill until it accumulates enough data to send, as determined by the amount of data the client wanted to receive (i.e. the size of the client's buffer) or the stream indicates that no more data exists. This is known as impedance matching and is described in subsequent sections. Note that for *active* streams the supplied buffer's **Recycle()** method may be called during this process when the framework no longer needs to use the buffer. When called, the buffer should either be deleted or reclaimed for future use. Additionally, the service may start to pre-fill a buffer to satisfy a future call to **GetBuff()** while the previous data is in transit to the client.
15) Using a new thread, the **SSI** framework matches the data with the request, **crP**, that asked for the data and reads the data into the buffer supplied by the client.
16) The framework invokes the request's **ProcessResponseData()** method. The application handles the data as needed and may repeat steps 9 and 10 if it needs more data. In any case, it must return to the caller.

17) At some point, the request is finished and the application invokes the request's **Finished()** method. If not all of the data has been received by the **SSI** framework, the server is notified to discard any remaining data. In any case, the framework removes all references to the request object which allows the application to delete the object. The request object may not be deleted until its **Finished()** method is called. On the server side, one of two poSSIbilities exist, as follows:
    a) If not all of the data has been transmitted to the client but a finished request is received, then for *active* streams the server-side **SSI** framework calls **Recycle()** on any buffers it still has control over. For all stream types, the framework invokes the corresponding request's **Finished()** method indicating that the request was cancelled.
    b) If the stream indicated that no more data exists and the remaining data has been sent to the client, then for *active* streams **SSI** framework calls **Recycle()** on any buffers it still has control over. For all stream types, the framework invokes the corresponding request's **Finished()** method indicating that the request was successfully completed.

    In either case, it invokes request object's **Finished()** method which invokes the **Finished()** method of the responder object bound to the request. That method can reclaim the stream object. Active streams are assured that either all buffers exported by the stream's **GetBuff()** method have already been recycled or will be upon return.

4.3.1 Active Streams

An active stream object is one that provides a buffer containing the data when requested to provide data. This means that the implementation of an active stream is responsible for buffer management. Active streams only make sense for a service provider (i.e. the server-side plug-in). Active streams are never created by the **SSI** client-side framework. A sample active stream derivation is shown below.

```
#include "XrdSsi/XrdSsiStream.hh"

class myStream : public XrdSsiStream
{public:

Buffer  *GetBuff(XrdSsiErrInfo &eInfo, int &dlen, bool &last);
•••
         myStream() : XrdSsiStream(XrdSsiStream::isActive) {…}
virtual ~myStream() {…}
•••
};
```

4.3.1.1 Active Stream Impedance Matching

Active streams pose a particular challenge since the **SSI** framework does not know how much data the active stream will deliver. Unless there is some indication in the request-response protocol, the client may not even know how much response data to expect. Consequently, clients typicaly provide a buffer of some size and ask the **SSI** framework to fill it to the extent possible.

When the server-side **SSI** framework initially receives a read request for the size equivalent to the size of the client's buffer, it asks the active stream for a buffer passing it the number of bytes needed to completely satisfy the read request. If the stream provides fewer bytes, the **SSI** framework subtracts the number of bytes returned from the original size, recycles the current buffer, and asks for another buffer of reduced size so that it can completely satisfy the request. This repeats until either the request is satisfied or the stream indicates that no more data remains.

If the active stream provides more data than is needed, the amount needed is used to satisfy the request and the remaining data is held (i.e. the buffer is not recycled). The data in the buffer is used to satisfy a subsequent request for data.

Subsequent requests for data either start anew or use data in an existing buffer to the extent possible. In this way, the server-side **SSI** framework matches unequal buffer sizes (i.e. impedance matches). Clearly, the most efficient transfer mode is where the active stream provides exactly the amount of data needed.

Impedance matching does not come without a latency cost. If the stream is slow in supplying data and the client's buffer is large then the client waits until the full buffer can be filled. This may cause the client to timeout and cancel the request. In the end, how an active stream supplies data is a trade-off between its speed and the amount of data requested by the client at one time. While small requests may be used to better match the speed of a stream, they come at the cost of more client-server interactions which incur additional network latency.

The good solution for slow streams is to pre-fill buffers ahead of client requests. The assumption is that it is rare that a request will be cancelled so as wasting the effort. The preceding stream diagram illustrates the appropriate places in the request-response sequence where pre-filling can be done with good results. Typically, two or three ready buffers should suffice.

The natural question is whether the initial buffer should be ready before posting the stream as a response to the request object. The answer really depends on the speed of the stream. If it is slower than the client turn-around then at least one buffer should be ready before posting the stream response. Otherwise, there is likely enough time to pre-fill at least one buffer afterwards. The general solution to this problem is to implement a consumer-produce algorithm with a fixed number of buffers.

4.3.2 Passive Streams

A passive stream is a stream that fills a caller supplied buffer. Passive stream need not manage any buffers as the caller always supplies one. Inherently, they are simpler to implement and provide the user of the stream complete freedom on how to manage the buffer space. Because of their simplicity and generality, only passive streams are used by the client-side **SSI** framework.

While passive streams provide much of the same capabilities as active stream they have one significant drawback; buffers cannot be pre-filled by a stream without significant data movement which, many times, is very expensive in terms of CPU utilization. Frequent and large movements of data also incur a large amount of memory contention that may cause significant latency in overall processing. Thus passive streams should only be used for data sources that can provide (i.e. fill a buffer) at reasonably high speeds. A typical derivation is shown below.

```
#include "XrdSsi/XrdSsiStream.hh"

class myStream : public XrdSsiStream
{public:

bool     SetBuff(XrdSsiRequest *rqstP, char *buff, int blen);
•••
         myStream() : XrdSsiStream(XrdSsiStream::isPassive)
                      {…}
virtual ~myStream() {…}
•••
};
```

4.3.2.1 Passive Stream Impedance Matching

While passive streams are relatively simple, the **SSI** framework still attempts to impedance match data availability from the stream with the amount of data requested by the client.

When the server-side **SSI** framework initially receives a read request for the size equivalent to the size of the client's buffer, it asks the passive stream to completely fill the buffer by passing it the number of bytes needed to completely satisfy the read request. If the stream provides fewer bytes, the **SSI** framework subtracts the number of bytes provided from the original size, and asks for another buffer fill of reduced size so that it can completely satisfy the request. This repeats until the either the request is satisfied or the stream indicates that no more data remains.

Impedance matching does not come without a latency cost. If the stream is slow in supplying data and the client's buffer is large then the client waits until the full buffer can be filled. This may cause the client to timeout and cancel the request. In the end, how an passive stream supplies data is a trade-off between its speed and the amount of data requested by the client at one time. While small requests may be used to better match the speed of a stream they come at the cost of more client-server interactions which incur additional network latency.

Because of the simplicity of passive stream, there is no efficient way to pre-fill buffers without incurring excessive memory-to-memory data movement which may cause excessive CPU usage or memory access latency.

5 Clustering SSI Servers

Since **SSI** executes in the **XRootD** framework, all **SSI** servers can be clustered using the native **XRootD** clustering services via the **cmsd** daemon. An **XRootD** cluster uses a manager node (also known as the redirector) to manage up to 64 server nodes. If your configuration has more than 64 servers, these are handled by additional daemons called supervisor nodes. Each supervisor can handle up to 64 nodes and there is no limit on the number of supervisors you may have. The details of **XRootD** clustering can be found in the Cluster Management Service Configuration Reference. In this chapter we focus on a simple clustered configuration and what needs to be done by your **SSI** server-side implementation to support it.

First, you need to understand how the **cmsd** locates resources in a cluster. The basic steps are as follows:

- A client connects to the **XRootD** daemon (hitherto called **xrootd**) that pairs with a manager **cmsd**. Here the **xrootd** handles data traffic while the **cmsd** is responsible for locating resources and monitoring the health of a cluster.
- The client then sends the resource name to the manager **xrootd**. Since the manager **xrootd** knows it is in a clustered environment; it asks its corresponding **cmsd** to locate the resource to be used by a request.
- If the **cmsd** has not seen the resource before, it sends a query to every **SSI** server node that potentially has the resource asking one or more to respond affirmatively if it can provide that resource.
- If no **SSI** servers respond then the client is told that the resource is not available.
- If one or more **SSI** servers respond that they have the resource, the **cmsd** chooses one of the **SSI** servers and tells the **xrootd** where to redirect the client to execute the request using the resource.
- The client then reconnects to the server that has the resource and asks it to execute the corresponding request. All subsequent interactions relative to the resource/request are directed to that server.

The above scenario means that the **cmsd** on the **SSI** server needs ask your plug in whether or not the resource is available using some kind of interface. This is done using the **QueryResource()** method in the **XrdSsiProvider** class. The steps that need to be taken are very similar to those required to create a provider for the **xrootd** daemon. The next section describes what needs to be done for the **cmsd**.

5.1 Define the Service Provider for the cmsd

Server-side services are provided via the **XrdSsiProvider** object. While the client-side has one that is built-in the server-side must define one. There are actually two types of providers:

- A **cmsd** provider that needs only to ascertain whether or not a resource is available on the node on which its **QueryResource()** method is called, and
- A **xrootd** provider that actually provides the service via the **GetService()** method. The provider still needs to be able to ascertain resource availability.

The lookup provider is used by **cmsd** to ascertain the availability of a resource on a particular server node. Since the **cmsd** never actually provides any service it *never* obtains a service object from the provider object. The lookup provider object is pointed to by the global pointer **XrdSsiProviderLookup** which you must define and set at library load time (i.e. it is a file level global static symbol).

The service provider is used by **xrootd** to actually process client requests. This was described in a previous section. While it can be the same service provider object; in most cases it makes sense to have two different objects provide the **QueryResource()** method as it usually it differs in actual implementation.

When your library is loaded, the **cmsd** locates the provider using the symbol **XrdSsiProviderLookup**. Initialization fails if the symbol cannot be found or if its value is nil.

The two methods that you must implement in your derived **XrdSsiProvider** object for the **cmsd** are:

- **Init()**, and
- **QueryResource()**

When a client issues a request with a resource, and the resource has not been used before, all servers indicating that they can provide the resource are asked via this method if they can actually provide the resource. The **SSI** framework then picks one of those servers to handle the client's requests relative to the resource. This effectively implements a real-time dynamic resource registration system.

The following code snippet shows how you would typically define the provider pointer at file level.

```
#include "XrdSsi/XrdSsiProvider.hh"

class MyLookupProvider : public XrdSsiProvider {•••};

XrdSsiProvider *XrdSsiProviderLookup = new MyLookupProvider;
```

Once the provider object is found, its **Init()** method is called. The method should initialize the object for its intended use. Below is a sample derivation of an **XrdSsiProvider** class for the **cmsd**.

```
#include "XrdSsi/XrdSsiProvider.hh"
#include "XrdSsi/XrdSsiService.hh"

class myLookUpProvider : public XrdSsiProvider
{
public:

// Init() is always called before any other method
//
bool           Init(XrdSsiLogger *logP,  XrdSsiCluster *clsP,
                    std::string   cfgFn, std::string    parms,
                    int           argc,  char         **argv

                   ) {••• initOK = true; // If all went well
                      return initOK;
                     }

// The QueryResource() method determine resource availability
//
XrdSsiProvider::
rStat          QueryResource(const char *rName,
                             const char *contact=0
                            );

               myLookUpProvider() : initOK(false);
virtual       ~myLookUpProvider() {}

private:

bool initOK;
};
```

6 Client Configuration

Generally, there is no need to configure anything on the client's side. The client side program simply links against **libXrdSsiLib.so** to obtain access to the client-side **SSI** framework. In some cases, you may want to change certain defaults at start-up time. These are done by calling certain static methods in the **XrdSsiProvider** class prior to obtaining any service objects. The various options that are available are described below.

6.1 Number of threads

You specify the maximum number of threads to use to handle **SSI** affairs by calling **XrdSsiProvider:: SetCBThreads()**.

1) Callback threads: The number of threads used for callbacks establish the maximum number of callbacks that may be active at the same time. The default is 300 but can be no more than 32,767.

2) Network threads: these drive callback threads. In practice, far fewer of these threads are needed and the default is 10% of the number of callback threads. The specifiable range is between 3 and 100.

Be aware that in **Linux**, the **nproc ulimit** determines the actual number of threads that can be used. Make sure that the **nproc** limit is not less than the total number of threads you want. Add at least 10% to the value for other background threads.

6.2 Default Timeouts

Various timeouts are used to detect error events. Most of these timeouts are set to infinity and error detection is performed in a time insensitive manner. You may wish to adjust these timeouts by calling **XrdSsiProvider:: SetTimeout()** based on what you are actually doing. The following table explains the timeouts.

Type	Default	Meaning of Timeout
connect_N	5	Number of times to retry a failed server connection
connect_T	120	Seconds to wait for a connection to complete before retrying
idleClose	∞	Seconds a socket may remain idle before it is closed
request_T	∞	Seconds to wait for a server interaction to complete
stream_T	∞	Seconds that a socket may be idle with outstanding requests

6.3 Request Timeout

Request timeouts may be specified on an individual basis by calling the protected **XrdSsiRequest::SetTimeOut()** method on the request object whose timeout you wush to set prior to calling **XrdSsiService::ProcessRequest()** with that request object. The default uses the **XrdSsiProvider::request_T** global setting.

7 Server Configuration

In order to use the server-side **SSI** framework you must configure an **XRootD** server. There are two modes: unclustered (i.e. stand-alone single server) and clustered (i.e. multiple servers clustered by a redirector or manager node).

The following sections show the minimal set[1] of **XRootD** directives needed to use the **SSI** framework for each type of configuration. Finally, the list of **SSI**-specific directives is explained. These may help you customize the **SSI** framework.

7.1 Resource Name Configuration

While resource names are arbitrary, by default they must start with **/tmp**; which is rather useless. You control the form of valid resource names using the **all.export** configuration directive. This directive is essentially *mandatory* in order to be able to use resources. The directive allows you to specify the leading characters of a valid resource name. There can be any number of these directives. For instance,

```
all.export /resource/ nolock r/w
```

Specifies that a valid resource must start with the sequence "/resource/" followed by a sequence of characters that includes Names are restricted to the following set of characters:

- Letters (upper or lower case),
- Digits (0-9), and
- Special characters: `!@#%^_-+=:./`

In general, paths may not contain shell meta-characters or imbedded spaces. Be aware that you should always specify the **nolock** and **r/w** options in the order shown; otherwise, requests will likely fail.

It is possible to lift most restrictions on resource names by specifying the following configuration directive:

```
all.export *? nolock r/w
```

[1] For a complete list of directives see the relevant reference manuals on **XRootD** and **cmsd**.

Resource names are not checked for validity but still may not contain imbedded blank characters. The added question makes **XRootD** scan for a question mark and split the resource into a name and a **CGI** string.

7.2 Unclustered XRootD SSI Configuration

```
# Tell XRootD to use only the SSI framework. If you wish to
# also use the filesystem features of XRootD then add the
# keyword default (i.e. xrootd.fslib libXrdSsi.so default).
#
xrootd.fslib libXrdSsi.so

# Turn off async processing as this does not work with SSI
#
xrootd.async off

# Declare the valid prefix for resource names. You can have
# as many of these directives as you wish, each specifying a
# different prefix (substitute the actual prefix for respfx).
# If you wish to use resource names without a leading slash,
# read the section describing resource name configuration.
#
all.export respfx nolock r/w

# Specify the resource lookup function to be used.
#
oss.statlib -2 libXrdSsi.so

# Specify the location of the shared library implementing
# you SSI service. See the SSI svclib directive for details.
#
ssi.svclib libpath
```

7.3 Clustered XRootD SSI Configuration

In addition to the directives specified for an unclustered configuration, you need to specify the additional directives shown below.

```
# Tell XRootD who the cluster manager is (a.k.a. redirector).
# Substitute for manhost the fully qualified DNS name of the
# node running the cluster manager. For manport the port
# number that it should use to listen for requests. Do so
# everywhere you see manhost and manport here.
#
all.manager manhost:manport

# Assign the appropriate role to SSI servers and the cluster
# manager. This is done using an if-else-fi clause. In this
# way the same configuration file can be used everywhere.
#
if manhost
all.role manager
else
all.role server
fi
```

7.4 Separating SSI and XRootD Resource Names

Recall that the **all.export** directive tells the **SSI** framework how to process resource names. It is likely you will need to specify this directive to establish a coherent naming convention for your resources. While there are numerous options only two are meaningful for the SSI framework, **nolock** and **r/w**; as previously described.

While resource names are typically used to identify **SSI** specific resources, the **SSI** framework allows you to also define traditional file system resources that can be used by clients using the same server. This is enabled using the **ssi.fspath** directive described under the **SSI** specific directives.

When you wish to enable an **xrootd** server to handle **SSI** requests as well as file system request, you need to alter the **xrootd.fslib** directive, add **ssi.fspath** directives, and the appropriate **all.export** directives, as shown below.

```
# Tell XRootD to handle SSI requests as well as regular XRootD
# file system requests (notice the addition of default).
#
xrootd.fslib libXrdSsi.so default

# Tell the SSI framework which resource name prefixes refer
# to actual file system requests. Substitute for fspfx a
# file system path prefix. There can be many ssi.fspath
# directives.
#
ssi.fspath fspfx

# Declare the valid prefix for resource names. You can have
# as many of these directives as you wish, each specifying a
# different prefix. Substitute the actual prefix for respfx
# You must also export the file system prefixes declared via
# the ssi.fspath directive. However, you must not specify the
# nolock option for these to prevent file corruption.
#
all.export respfx nolock r/w

all.export fspfx appropriate_options

# Continue with the directives specified in the previous
# sections for unclustered or clustered configurations.
```

7.5 SSI Specific Directives

All **SSI**-specific directives start with the prefix "**ssi.**" to differentiate them from other types of directives.

7.5.1 fspath

```
ssi.fspath fspfx
```

Function

Specify the resource name prefix that routes an **SSI** request to the **XRootD** file system plug-in.

Parameters

fspfx The resource name prefix, when seen, is to route the request to the file system plug-in instead of the **SSI** service.

Default

There is no default, see the notes for more information.

Notes

1) If you enable the **XRootD** file system plug-in to function alongside the **SSI** framework then you need to specify which resource name prefixes actually refer to the file system and not the **SSI** service. When a resource name with the matching prefix is encountered, the request is routed to the file system plug-in.
2) Each *fspfx* specified with the **fspath** directive must also be specified using the all.export directive. However, you must *not* specify the **nolock** option for these exports.

Example

```
ssi.fspath /tmp
all.export /tmp
```

7.5.2 opts

```
ssi.opts [authdns] [detreqok] [maxrsz rsz[k | m | g]]
         [requests rnum] [respwt sec]
```

Function

Specify the resource name prefix that routes an **SSI** request to the **XRootD** file system plug-in.

Parameters

authdns

Specifies that the client's host name be fully resolved in the supplied authentication information. By default, the host identification is determined by the **XRootD xrd.network** configuration option. Hence, the host identification may be a host name or an IP address.

detreqok

Specifies that client's may execute detached requests. By default, detached requests are disallowed.

maxrsz *rsz*

Specifies the maximum size of a valid request. Specify for *rsz* the largest possible request size. The *rsz* can be suffixed by **k**, **m**, or **g** to indicate **k**ilo-, **m**ega-, or **g**iga-bytes; respectively. The default is 2m. See the next section on how to better optimize values greater than two megabytes.

requests *rval*

Specifies the maximum number of request objects to hold in reserve for future requests. Specify for *rval* a number greater than 0 but no more than 64k. The default is 256.

respwt *sec*

Specifies the number of seconds the client should be asked to wait for a response when a response is not ready. The *sec* can be suffixed by **d**, **h**, or **m**, or **s** to indicate **d**ays, **m**inutes, or **h**ours; respectively. The default is 24855d.

Default

```
ssi.opts maxrsz 2m requests 256 respwt 24855d
```

Example

```
ssi.opts requests 512
```

7.5.2.1 Optimizing Large Request Sizes

The default **XRootD** transfer unit is set to 2 megabytes. If you specify an **SSI** request size that is greater than 2MB and the **SSI** framework receives a request that is greater than 2MB then the request data must be copied in 2MB units into a contiguous area of storage before presenting the request to the service. This can represent significant overhead if many requests are larger than 2MB.

You can eliminate the overhead if you make the **XRootD** transfer unit match the maximum **SSI** request size. You do this by via the **xrd.buffers** directive, shown below.

```
xrd.buffers maxbsz ssi_rsz[k | m | g]
```

Other options for this directive are documented in the "**Xrd/XRootD** *Configuration Reference*".

7.5.3 svclib (*required*)

```
ssi.svclib lib [parms]
```

Function

Specify the shared library that implements the SSI service.

Parameters

lib The path to the shared library that contains the code the implements the protocol.

parms Parameters to be passed to the service initialization method at load time.

Default

There is no default; this is a required directive.

Example

```
ssi.svclib libService.so
```

7.5.4 trace

```
ssi.trace [-]option

option:  {all | debug | off} [[-]option]
```

Function

Specify execution tracing options.

Parameters

option Specifies the tracing level. One or more options may be specified. The specifications are cumulative and processed left to right. Each option may be optionally prefixed by a minus sign to turn off the setting. Valid options are:

all	selects all possible trace levels other than debug
debug	adds additional tracing for debugging purposes
off	traces nothing

Defaults

Tracing is disabled.

Notes

1) All tracing is forcibly enabled when the daemon is invoked with the **–d** option.
2) All previous trace settings are discarded when **off** is encountered.

Example

```
ssi.trace all
```

8 Managing Resources in a Cluster

When the **XrdSsiProvider::Init()** method is called in a clustered environment, it is passed a pointer to an **XrdSsiCluster** object. This object is used to manage resources within cluster. It is critical that resources be properly managed for efficient execution of client requests in the cluster. The two main actions using the cluster object are:

1. Register or unregister the presence of a resource name at a node.
2. Request suspension or resumption of service.

8.1 Registering and Unregistering Resource Names

Recall that when the cmsd looks nodes that can provide a resource name that it has not yet seen, it asks each node whether or not it has the particular resource. Nodes responding that they have the resource are eligible to receive client requests for that resource. This mechanism is used to automatically register (affirmative response) and unregister (no response) resource names relative to the set of nodes in the cluster. All of this is driven by calling **XrdSsiProvider::QueryResource()**.

Since automatic registration is specific to a point in time, the cluster object has methods that must be used by a node to inform the cluster manager of changes in the registration status of its resources. Specifically,

Added(const char ****name*, bool** *pend***=false)**

must be called whenever a new resource becomes available or has changed its pending status since the last time the node was queried about the resource name. Since it is inconvenient to track whether or not a node was asked about the *name*, the **Added()** method may be called at any time whether or not the node was queried about the resource. This allows registrations to be current and accurate.

Removed(const char ****name*)**

must be called whenever a new resource becomes unavailable since the last time the node was queried about the resource name. Since it is inconvenient to track whether or not a node was asked about the *name*, the **Removed()** method may be called at any time whether or not the node was queried about the resource. This allows registrations to be current and accurate.

Failure to properly use these two methods may cause clients to be sent to the wrong nodes or being told a resource does not exist even when it does. While the former is auto-correcting with an added latency cost the latter is not correctable and the client will normally declare a fatal error.

8.2 Suspending and Resuming Service

A node has control of whether or not it is willing to accept client requests. At initial start-up the cluster manager assumes all nodes in the cluster can accept client requests unless told otherwise. A node can control it status by using the following **XrdSsiCluster** methods:

Suspend(bool *perm***=false)**

may be called to prohibit clients from being sent to the issuing node in the future. When perm is true, this status persists across node restarts. Otherwise, the suspension is cancelled after a server at the node restarts.

Resume(bool *perm***=true)**

may be called to resume service after a previous Suspend() call. When called with perm equal to true, any permanent suspension undone. Passing a value of false, maintains the previous suspension status upon server restart.

Permanent suspensions are meant to easily implement maintenance mode. Should a server's node enter maintenance the server can be permanently suspended to make sure that an inadvertent restart does not resume service before the completion of maintenance.

Since suspensions are time driven events, it is possible that client requests have been directed to the server before the suspension is acknowledged by the cluster manager. So, it is normal to still get several requests after calling **Suspend()**. In this case, the server may redirect these requests back to the cluster manager for reprocessing.

Suspend() and **Resume()** can be used to manage the load at the server. Three special helper methods are available to make it easier to do so. These **XrdSsiCluster** methods are based on the concept of unit of service and are used as follows:

Resource(int *n*)

is used to declare *n* arbitrary units of available service. While it is normally called once, it can be called any number of times. Each invocation returns the resource unit value that was previously established.

Reserve(int *n*=1)

is used to declare that the server has undertaken a task that will use *n* units of service. The default is one unit. The amount is deducted from the available service units and should the remainder fall below 1; **Suspend()** is automatically called to temporarily suspend service.

Release(int *n*=1)

is used to declare that the server has completed a task that required the *n* units of service. The default is one unit. The amount is added to the available service units. If the amount available transitions from a non-positive amount to a positive value; **Resume()** is automatically called to allow new client requests to be sent to the server.

9 Starting the SSI Server

Starting an **SSI** server is equivalent to starting an xrootd in stand-alone node or an xrootd-cmsd pair in a clustered node. There are numerous command line options for starting each type of server. Refer to the **Xrd/Xroot**d and **cmsd** reference manuals for details on these options. With few exceptions, they are the same for **xrootd** and **cmsd** daemons. There is one common option that deserves special treatment here. The option allows you to pass command line arguments to the **XrdSsiProvider:: Init()** method. When starting a server-side daemon you may specify command line arguments to be passed to the **SSI** plug-in as follows.

{**xrootd** | **cmsd**} [*options*] [*arguments*] **-+xrdssi** *options_arguments*

The *options_arguments* tokens are passed to the **XrdSsiProvider::Init()** method via the *argv* parameter with *argc* set to the actual number of arguments. The **argv[0]** token is the same as the one passed to the daemon (i.e. executable path). The *options_arguments* tokens end either at the end of the command line or when another "**-+**" option is encountered. The **-+xrdssi** option must appear after all of the **xrootd**/**cmsd** specific options and arguments have been specified on the command line.

10 Document Change History

9 Feb 2016

- Document produced.